

EK690671357US

1

Docket No. AUS990853US1

**METHOD AND SYSTEM FOR APPORTIONING CHANGES IN METRIC
VARIABLES IN AN SYMMETRIC MULTIPROCESSOR (SMP)
ENVIRONMENT**

CROSS-REFERENCE TO RELATED APPLICATIONS

This application is related to the following copending and commonly assigned applications entitled "SYSTEM AND METHOD FOR PROVIDING TRACE INFORMATION REDUCTION", U.S. Application Serial Number 08/989,725, now issued as U.S. Patent Number 6,055,492, filed on December 12, 1997, currently pending, "A METHOD AND APPARATUS FOR STRUCTURED PROFILING OF DATA PROCESSING SYSTEMS AND APPLICATIONS", U.S. Application Serial Number 09/052,329, Attorney Docket Number AT9-98-074, currently pending, filed on March 31, 1998, "A METHOD AND APPARATUS FOR STRUCTURED MEMORY ANALYSIS OF DATA PROCESSING SYSTEMS AND APPLICATIONS", U.S. Application Serial Number 09/052,331, Attorney Docket Number AT9-98-075, currently pending, filed on March 31, 1998, and "METHOD AND APPARATUS FOR PROFILING PROCESSES IN A DATA PROCESSING SYSTEM", U.S. Application Serial Number 09/177,031, Attorney Docket Number AT9-98-295, currently pending, filed on October 22, 1998, "PROCESS AND SYSTEM FOR MERGING TRACE DATA FOR PRIMARILY INTERPRETED METHODS", U.S. Application Serial Number 09/343,439, Attorney Docket Number AT9-98-849, currently pending, filed on June 30, 1999; "METHOD AND SYSTEM FOR MERGING EVENT-BASED DATA AND SAMPLED DATA INTO POSTPROCESSED TRACE OUTPUT", U.S. Application Serial Number 09/343,438, Attorney Docket Number AT9-98-850, currently pending, filed June 30, 1999; "METHOD AND SYSTEM FOR SHADOW HEAP MEMORY LEAK DETECTION AND OTHER HEAP ANALYSIS IN AN OBJECT-ORIENTED

0000000000 - 0000000000

Docket No. AUS990853US1

ENVIRONMENT DURING REAL-TIME TRACE PROCESSING", U.S. Application Serial Number _____, Attorney Docket Number AUS990853US1, filed _____; "METHOD AND SYSTEM FOR TRACING PROFILING INFORMATION IN AN APPLICATION USING PER THREAD METRIC VARIABLES WITH REUSED KERNEL THREADS", U.S. Application Serial Number - _____, Attorney Docket Number AUS000055US1, filed _____; "METHOD AND SYSTEM FOR TRACING PROFILING INFORMATION USING PER THREAD METRIC VARIABLES WITH REUSED KERNEL THREADS", U.S. Application Serial Number _____, Attorney Docket Number AUS000057US1, filed _____; and "METHOD AND SYSTEM FOR SMP PROFILING USING SYNCHRONIZED OR NONSYNCHRONIZED METRIC VARIABLES WITH SUPPORT ACROSS MULTIPLE SYSTEMS", U.S. Application Serial Number _____, Attorney Docket Number AUS000129US1, filed _____.

00000000-0000-0000-0000-000000000000

BACKGROUND OF THE INVENTION**1. Technical Field:**

The present invention relates to an improved data processing system and, in particular, to a method and apparatus for optimizing performance in a data processing system. Still more particularly, the present invention provides a method and apparatus for a software program development tool for enhancing performance of a software program through software profiling.

2. Description of Related Art:

In analyzing and enhancing performance of a data processing system and the applications executing within the data processing system, it is helpful to know which software modules within a data processing system are using system resources. Effective management and enhancement of data processing systems requires knowing how and when various system resources are being used. Performance tools are used to monitor and examine a data processing system to determine resource consumption as various software applications are executing within the data processing system. For example, a performance tool may identify the most frequently executed modules and instructions in a data processing system, or may identify those modules, which allocate the largest amount of memory or perform the most I/O requests. Hardware performance tools may be built into the system or added at a later point in time. Software performance tools also are useful in data processing systems, such as personal computer systems, which typically do not contain many, if any, built-in hardware performance tools.

00000000000000000000000000000000

One known software performance tool is a trace tool. A trace tool may use more than one technique to provide trace information that indicates execution flows for an executing program. One technique keeps track of particular sequences of instructions by logging certain events as they occur, so-called event-based profiling technique. For example, a trace tool may log every entry into, and every exit from, a module, subroutine, method, function, or system component. Alternately, a trace tool may log the requester and the amounts of memory allocated for each memory allocation request. Typically, a time-stamped record is produced for each such event. Corresponding pairs of records similar to entry-exit records also are used to trace execution of arbitrary code segments, starting and completing I/O or data transmission, and for many other events of interest.

In order to improve performance of code generated by various families of computers, it is often necessary to determine where time is being spent by the processor in executing code, such efforts being commonly known in the computer processing arts as locating "hot spots." Ideally, one would like to isolate such hot spots at the instruction and/or source line of code level in order to focus attention on areas, which might benefit most from improvements to the code.

Another trace technique involves periodically sampling a program's execution flows to identify certain locations in the program in which the program appears to spend large amounts of time. This technique is based on the idea of periodically interrupting the application or data processing system execution at regular intervals, so-called sample-based profiling. At each interruption,

information is recorded for a predetermined length of time or for a predetermined number of events of interest. For example, the program counter of the currently executing thread, which is a process that is part of the larger program being profiled, may be recorded during the intervals. These values may be resolved against a load map and symbol table information for the data processing system at post-processing time, and a profile of where the time is being spent may be obtained from this analysis.

For example, isolating such hot spots to the instruction level permits compiler writers to find significant areas of suboptimal code generation at which they may thus focus their efforts to improve code generation efficiency. Another potential use of instruction level detail is to provide guidance to the designer of future systems. Such designers employ profiling tools to find characteristic code sequences and/or single instructions that require optimization for the available software for a given type of hardware.

When profiling includes gather profiling information at the processor level, the profiler must rely on the operating system for the profile information. The profiler cannot know if the change in a global variable returned by the operating system is the change from two or more processors concurrently executing code. A non-SMP enabled profiler would capture the values at thread switches and at entries and exits and apply the change to the method on the specified thread.

In the SMP case, if two processors update the same global variable simultaneously, it is impossible to apportion the change in a global variable between two

processors. Individual values from each SMP processor are impossible to determine because an SMP-enabled profiler would capture values for processors from the operating system kernel concurrently. If two processors update the same global variable concurrently, it cannot be determined how much of the change is due to a thread on a first processor and how much of the change is due to a second thread on a second processor.

Therefore, it would be advantageous to provide a system in which the contribution from each processor on the change of a variable can ascertained.

09612345 - 070608

SUMMARY OF THE INVENTION

A method and system for monitoring performance of a program using global metric variables to provide the support in a symmetric multiprocessor (SMP) system. In one embodiment a Java virtual machine (JVM) either calls the profiler whenever bytes are allocated or provides an interface to allow the profiler to determine the value of the change in the metric for the current thread. The profiler then applies the changes to a metric for the current thread. In another embodiment, per processor data areas are maintained for storing per processor metric values. Whenever a thread switch occurs or there is a request for the metric on a specified thread, an operating system kernel updates the thread level metric values with changes in the values per processor metrics.

卷之三

BRIEF DESCRIPTION OF THE DRAWINGS

The novel features believed characteristic of the invention are set forth in the appended claims. The invention itself, however, as well as a preferred mode of use, further objectives and advantages thereof, will best be understood by reference to the following detailed description of an illustrative embodiment when read in conjunction with the accompanying drawings, wherein:

Figure 1 is an illustration depicting a distributed data processing system in which the present invention may be implemented;

Figure 2A-B are block diagrams depicting a data processing system in which the present invention may be implemented;

Figure 3A is a block diagram depicting the relationship of software components operating within a computer system that may implement the present invention;

Figure 3B is a block diagram depicting a Java virtual machine in accordance with a preferred embodiment of the present invention;

Figure 4 is a block diagram depicting components used to profile processes in a data processing system;

Figure 5 is an illustration depicting various phases in profiling the active processes in an operating system;

Figure 6 is a flowchart depicting a process used by a trace program for generating trace records from processes executing on a data processing system;

Figure 7 is a flowchart depicting a process used in a system interrupt handler trace hook;

Figure 8 is a diagram depicting the call stack

00000000000000000000000000000000

containing stack frames;

Figure 9 is an illustration depicting a call stack sample;

Figure 10A is a diagram depicting a program execution sequence along with the state of the call stack at each function entry/exit point;

Figure 10B is a diagram depicting a particular timer based sampling of the execution flow depicted in **Figure 10A**;

Figure 10C-D are time charts providing an example of the types of time for which the profiling tool accounts;

Figure 11A is a diagram depicting a tree structure generated from sampling a call stack;

Figure 11B is a diagram depicting an event tree, which reflects call stacks observed during system execution;

Figure 12 is a table depicting a call stack tree;

Figure 13 is a flow chart depicting a method for building a call stack tree using a trace text file as input;

Figure 14 is a flow chart depicting a method for building a call stack tree dynamically as tracing is taking place during system execution;

Figure 15A is a flowchart depicting a process for creating a call stack tree structure from call stack unwind records in a trace file;

Figure 15B is a flowchart depicting a process for identifying functions from an address obtained during sampling;

Figure 16 is a diagram depicting a record generated using the processes of the present invention;

Figure 17 is a diagram depicting another type of report that may be produced to show the calling structure between routines shown in **Figure 12**;

Figure 18 is a table depicting a report generated from a trace file containing both event-based profiling information (method entry/exits) and sample-based profiling information (stack unwinds);

Figure 19 is a table depicting major codes and minor codes that may be employed to instrument modules for profiling;

Figure 20 is a block diagram depicting the relationship between a profiler and other software components in a data processing system capable of application level thread oriented process for tracking metrics on a per thread basis;

Figure 21A is a flowchart illustrating a process for initializing the profiler for performing the application level thread-oriented approach for tracking metrics (M) in accordance with a preferred embodiment of the present invention;

Figure 21B is a flowchart illustrating the process for allocating new data areas in response to a receiving an entry or exit thread event on new thread (t_{new}) for the previously initialized metrics (M);

Figure 22 is a flowchart depicting a process for tracking a individual metric in response to a metric event in accordance with a preferred embodiment of the present invention;

Figure 23 is a flowchart depicting a process for tracking metrics in response to a method entry or exit event occurring on current thread (t) in accordance with

a preferred embodiment of the present invention;

Figure 24 is a flowchart depicting a simplified application programming interface (API) for retrieving metric values in accordance with a preferred embodiment of the present invention;

Figure 25 is a diagram illustrating the relationship of the Jvm, profiler and an operating system kernel for tracking metrics using an operating system kernel level per processor based approach in accordance with a preferred embodiment of the present invention;

Figure 26A is a flowchart depicting the initialization process for tracking metrics on a per processor basis in accordance with a preferred embodiment of the present invention;

Figure 26B is a flowchart depicting a process for initializing the operating system kernel in response to the kernel recognizing a new thread in accordance with a preferred embodiment of the present invention;

Figure 26C is a flowchart depicting a process for initializing the profiler in response to the profiler recognizing a new thread in accordance with a preferred embodiment of the present invention;

Figure 27 is a flowchart depicting a process for the operating system kernel updating a per processor metric variable being tracked, in response to a Jvm request in accordance with a preferred embodiment of the present invention;

Figure 28 is a flowchart depicting a process for the Jvm directly updating the value of a per processor metric variable in accordance with a preferred embodiment of the present invention;

Figure 29 is a flowchart depicting a process for the

operating system kernel updating the value of a per processor metric variable in response to a request from the Jvm in accordance with a preferred embodiment of the present invention;

Figure 30 is a flowchart depicting the operating system kernel updating a metric variable value in response to a thread dispatch event in accordance with a preferred embodiment of the present invention; and

Figure 31 is a flowchart depicting the process for updating metric variable values in response to a method entry or exit event in accordance with a preferred embodiment of the present invention.

DETAILED DESCRIPTION OF A PREFERRED EMBODIMENT

With reference now to the figures, and in particular with reference to **Figure 1**, a pictorial representation of a distributed data processing system in which the present invention may be implemented is depicted.

Distributed data processing system 100 is a network of computers in which the present invention may be implemented. Distributed data processing system 100 contains a network 102, which is the medium used to provide communications links between various devices and computers connected together within distributed data processing system 100. Network 102 may include permanent connections, such as wire or fiber optic cables, or temporary connections made through telephone connections.

In the depicted example, a server 104 is connected to network 102 along with storage unit 106. In addition, clients 108, 110, and 112 also are connected to a network 102. These clients 108, 110, and 112 may be, for example, personal computers or network computers. For purposes of this application, a network computer is any computer, coupled to a network, which receives a program or other application from another computer coupled to the network. In the depicted example, server 104 provides data, such as boot files, operating system images, and applications to clients 108-112. Clients 108, 110, and 112 are clients to server 104. Distributed data processing system 100 may include additional servers, clients, and other devices not shown. In the depicted example, distributed data processing system 100 is the Internet with network 102 representing a worldwide

collection of networks and gateways that use the TCP/IP suite of protocols to communicate with one another. At the heart of the Internet is a backbone of high-speed data communication lines between major nodes or host computers, consisting of thousands of commercial, government, educational, and other computer systems, that route data and messages. Of course, distributed data processing system 100 also may be implemented as a number of different types of networks, such as, for example, an Intranet or a local area network.

Figure 1 is intended as an example, and not as an architectural limitation for the processes of the present invention.

With reference now to **Figure 2A**, a block diagram of a data processing system which may be implemented as a server, such as server 104 in **Figure 1**, is depicted in accordance to the present invention. Data processing system 200 may be a symmetric multiprocessor (SMP) system including a plurality of processors 202 and 204 connected to system bus 206. Alternatively, a single processor system may be employed. Also connected to system bus 206 is memory controller/cache 208, which provides an interface to local memory 209. I/O Bus Bridge 210 is connected to system bus 206 and provides an interface to I/O bus 212. Memory controller/cache 208 and I/O Bus Bridge 210 may be integrated as depicted.

Peripheral component interconnect (PCI) bus bridge **214** connected to I/O bus **212** provides an interface to PCI local bus **216**. A modem **218** may be connected to PCI local bus **216**. Typical PCI bus implementations will support four PCI expansion slots or add-in connectors.

Communications links to network computers 108-112 in **Figure 1** may be provided through modem 218 and network adapter 220 connected to PCI local bus 216 through add-in boards.

Additional PCI bus bridges 222 and 224 provide interfaces for additional PCI buses 226 and 228, from which additional modems or network adapters may be supported. In this manner, server 200 allows connections to multiple network computers. A memory mapped graphics adapter 230 and hard disk 232 may also be connected to I/O bus 212 as depicted, either directly or indirectly.

Those of ordinary skill in the art will appreciate that the hardware depicted in **Figure 2A** may vary. For example, other peripheral devices, such as optical disk drive and the like also may be used in addition or in place of the hardware depicted. The depicted example is not meant to imply architectural limitations with respect to the present invention.

The data processing system depicted in **Figure 2A** may be, for example, an IBM RISC/System 6000 system, a product of International Business Machines Corporation in Armonk, New York, running the Advanced Interactive Executive (AIX) operating system.

With reference now to **Figure 2B**, a block diagram of a data processing system in which the present invention may be implemented is illustrated. Data processing system 250 is an example of a client computer. Data processing system 250 employs a peripheral component interconnect (PCI) local bus architecture. Although the depicted example employs a PCI bus, other bus architectures such as Micro Channel and ISA may be used.

Processor **252** and main memory **254** are connected to PCI local bus **256** through PCI Bridge **258**. PCI Bridge **258** also may include an integrated memory controller and cache memory for processor **252**. Additional connections to PCI local bus **256** may be made through direct component interconnection or through add-in boards. In the depicted example, local area network (LAN) adapter **260**, SCSI host bus adapter **262**, and expansion bus interface **264** are connected to PCI local bus **256** by direct component connection. In contrast, audio adapter **266**, graphics adapter **268**, and audio/video adapter (A/V) **269** are connected to PCI local bus **266** by add-in boards inserted into expansion slots. Expansion bus interface **264** provides a connection for a keyboard and mouse adapter **270**, modem **272**, and additional memory **274**. SCSI host bus adapter **262** provides a connection for hard disk drive **276**, tape drive **278**, and CD-ROM **280** in the depicted example. Typical PCI local bus implementations will support three or four PCI expansion slots or add-in connectors.

An operating system runs on processor **252** and is used to coordinate and provide control of various components within data processing system **250** in **Figure 2B**. The operating system may be a commercially available operating system such as JavaOS for BusinessTM or OS/2TM, which are available from International Business Machines CorporationTM. JavaOS is loaded from a server on a network to a network client and supports Java programs and applets. A couple of characteristics of JavaOS that are favorable for performing traces with stack unwinds, as described below, are that JavaOS does not support

00000000000000000000000000000000
paging or virtual memory. An object-oriented programming system such as Java may run in conjunction with the operating system and may provide calls to the operating system from Java programs or applications executing on data processing system 250. Instructions for the operating system, the object-oriented operating system, and applications or programs are located on storage devices, such as hard disk drive 276 and may be loaded into main memory 254 for execution by processor 252. Hard disk drives are often absent and memory is constrained when data processing system 250 is used as a network client.

Those of ordinary skill in the art will appreciate that the hardware in **Figure 2B** may vary depending on the implementation. For example, other peripheral devices, such as optical disk drives and the like may be used in addition to or in place of the hardware depicted in **Figure 2B**. The depicted example is not meant to imply architectural limitations with respect to the present invention. For example, the processes of the present invention may be applied to a multiprocessor data processing system.

The present invention provides a process and system for profiling software applications. Although the present invention may operate on a variety of computer platforms and operating systems, it may also operate within a Java runtime environment. Hence, the present invention may operate in conjunction with a Java virtual machine (JVM) yet within the boundaries of a Jvm as defined by Java standard specifications. In order to provide a context for the present invention, portions of the operation of a Jvm according to Java specifications

are herein described.

With reference now to **Figure 3A**, a block diagram illustrates the relationship of software components operating within a computer system that may implement the present invention. Java-based system **300** contains platform specific operating system **302** that provides hardware and system support to software executing on a specific hardware platform. Jvm **304** is one software application that may execute in conjunction with the operating system. Jvm **304** provides a Java run-time environment with the ability to execute Java application or applet **306**, which is a program, servlet, or software component written in the Java programming language. The computer system in which Jvm **304** operates may be similar to data processing system **200** or computer **100** described above. However, Jvm **304** may be implemented in dedicated hardware on a so-called Java chip, Java-on-silicon, or Java processor with an embedded picoJava core.

At the center of a Java run-time environment is the Jvm, which supports all aspects of Java's environment, including its architecture, security features, mobility across networks, and platform independence.

The Jvm is a virtual computer, i.e. a computer that is specified abstractly. The specification defines certain features that every Jvm must implement, with some range of design choices that may depend upon the platform on which the Jvm is designed to execute. For example, all Jvms must execute Java bytecodes and may use a range of techniques to execute the instructions represented by the bytecodes. A Jvm may be implemented completely in software or somewhat in hardware. This flexibility

allows different Jvms to be designed for mainframe computers and PDAs.

The Jvm is the name of a virtual computer component that actually executes Java programs. Java programs are not run directly by the central processor but instead by the Jvm, which is itself a piece of software running on the processor. The Jvm allows Java programs to be executed on a different platform as opposed to only the one platform for which the code was compiled. Java programs are compiled for the Jvm. In this manner, Java is able to support applications for many types of data processing systems, which may contain a variety of central processing units and operating systems architectures. To enable a Java application to execute on different types of data processing systems, a compiler typically generates an architecture-neutral file format. The compiled code is executable on many processors, given the presence of the Java run-time system. The Java compiler generates bytecode instructions that are nonspecific to a particular computer architecture. A bytecode is a machine independent-code generated by the Java compiler and executed by a Java interpreter. A Java interpreter is part of the Jvm that alternately decodes and interprets a bytecode or bytecodes. These bytecode instructions are designed to be easy to interpret on any computer and easily translated on the fly into native machine code. Byte codes may be translated into native code by a just-in-time compiler or JIT.

A Jvm must load class files and execute the bytecodes within them. The Jvm contains a class loader, which loads class files from an application and the class files from the Java application programming interfaces

00000000000000000000000000000000

(APIs), which are needed by the application. The execution engine that executes the bytecodes may vary across platforms and implementations.

One type of software-based execution engine is a just-in-time compiler. With this type of execution, the bytecodes of a method are compiled to native machine code upon successful fulfillment of some type of criteria for jitting a method. The native machine code for the method is then cached and reused upon the next invocation of the method. The execution engine may also be implemented in hardware and embedded on a chip so that the Java bytecodes are executed natively. Jvms usually interpret bytecodes, but Jvms may also use other techniques, such as just-in-time compiling, to execute bytecodes.

Interpreting code provides an additional benefit. Rather than instrumenting the Java source code, the interpreter may be instrumented. Trace data may be generated via selected events and timers through the instrumented interpreter without modifying the source code. Profile instrumentation is discussed in more detail further below.

When an application is executed on a Jvm that is implemented in software on a platform-specific operating system, a Java application may interact with the host operating system by invoking native methods. A Java method is written in the Java language, compiled to bytecodes, and stored in class files. A native method is written in some other language and compiled to the native machine code of a particular processor. Native methods are stored in a dynamically linked library whose exact form is platform specific.

With reference now to **Figure 3B**, a block diagram of a Jvm

is depicted in accordance with a preferred embodiment of the present invention. Jvm 350 includes a class loader subsystem 352, which is a mechanism for loading types, such as classes and interfaces, given fully qualified names. Jvm 350 also contains runtime data areas 354, execution engine 356, native method interface 358, and memory management 374. Execution engine 356 is a mechanism for executing instructions contained in the methods of classes loaded by class loader subsystem 352. Execution engine 356 may be, for example, Java interpreter 362 or just-in-time compiler 360. Native method interface 358 allows access to resources in the underlying operating system. Native method interface 358 may be, for example, a Java native interface. Runtime data areas 354 contain native method stacks 364, Java stacks 366, PC registers 368, method area 370, and heap 372. These different data areas represent the organization of memory needed by Jvm 350 to execute a program.

Java stacks **366** are used to store the state of Java method invocations. When a new thread is launched, the Jvm creates a new Java stack for the thread. The Jvm performs only two operations directly on Java stacks: it pushes and pops frames. A thread's Java stack stores the state of Java method invocations for the thread. The state of a Java method invocation includes its local variables, the parameters with which it was invoked, its return value, if any, and intermediate calculations. Java stacks are composed of stack frames. A stack frame contains the state of a single Java method invocation. When a thread invokes a method, the Jvm pushes a new

frame onto the Java stack of the thread. When the method completes, the Jvm pops the frame for that method and discards it. The Jvm does not have any registers for holding intermediate values; any Java instruction that requires or produces an intermediate value uses the stack for holding the intermediate values. In this manner, the Java instruction set is well defined for a variety of platform architectures.

PC registers **368** are used to indicate the next instruction to be executed. Each instantiated thread gets its own PC register (program counter) and Java stack. If the thread is executing a Jvm method, the value of the PC register indicates the next instruction to execute. If the thread is executing a native method, then the contents of the PC register are undefined. Native method stacks **364** store the state of invocations of native methods. The state of native method invocations is stored in an implementation-dependent way in native method stacks, registers, or other implementation-dependent memory areas. In some Jvm implementations, native method stacks **364** and Java stacks **366** are combined.

Method area **370** contains class data while heap **372** contains all instantiated objects. The Jvm specification strictly defines data types and operations. Most JVMs choose to have one method area and one heap, each of which are shared by all threads running inside the JVM. When the Jvm loads a class file, it parses information about a type from the binary data contained in the class file. It places this type information into the method area. Each time a class instance or array is created, the memory for the new object is allocated from heap **372**.

Jvm 350 includes an instruction that allocates memory space within the memory for heap 372 but includes no instruction for freeing that space within the memory. Memory management 374 in the depicted example manages memory space within the memory allocated to heap 370. Memory management 374 may include a garbage collector, which automatically reclaims memory used by objects that are no longer referenced. Additionally, a garbage collector also may move objects to reduce heap fragmentation.

The processes within the following figures provide an overall perspective of the many processes employed within the present invention: processes that generate event-based profiling information in the form of specific types of records in a trace file; processes that generate sample-based profiling information in the form of specific types of records in a trace file; processes that read the trace records to generate more useful information to be placed into profile reports; and processes that generate the profile reports for the user of the profiling utility.

With reference now to **Figure 4**, a block diagram depicts components used to profile processes in a data processing system. A trace program 400 is used to profile processes 402. Trace program 400 may be used to record data upon the execution of a hook, which is a specialized piece of code at a specific location in a routine or program in which other routines may be connected. Trace hooks are typically inserted for the purpose of debugging, performance analysis, or enhancing functionality. These trace hooks are employed to send trace data to trace program 400, which stores the trace

data in buffer **404**. The trace data in buffer **404** may be stored in a file for post-processing. With Java operating systems, the present invention employs trace hooks that aid in identifying methods that may be used in processes **402**. In addition, since classes may be loaded and unloaded, these changes may also be identified using trace data. This is especially relevant with "network client" data processing systems, such as those that may operate under JavaOS, since classes and jitted methods may be loaded and unloaded more frequently due to the constrained memory and role as a network client.

With reference now to **Figure 5**, a diagram depicts various phases in profiling the processes active in an operating system. Subject to memory constraints, the generated trace output may be as long and as detailed as the analyst requires for the purpose of profiling a particular program.

An initialization phase **500** is used to capture the state of the client machine at the time tracing is initiated. This trace initialization data includes trace records that identify all existing threads, all loaded classes, and all methods for the loaded classes. Records from trace data captured from hooks are written to indicate thread switches, interrupts, and loading and unloading of classes and jitted methods. Any class, which is loaded, has trace records that indicate the name of the class and its methods. In the depicted example, four byte IDs are used as identifiers for threads, classes, and methods. These IDs are associated with names output in the records. A record is written to indicate when all of the start up information has been written.

00000000-0000-0000-0000-000000000000

Next, during the profiling phase 502, trace records are written to a trace buffer or file. Trace records may originate from two types of profiling actions—event-based profiling and sample-based profiling. In the present invention, the trace file may have a combination of event-based records, such as those that may originate from a trace hook executed in response to a particular type of event, e.g., a method entry or method exit, and sample-based records, such as those that may originate from a stack walking function executed in response to a timer interrupt, e.g., a stack unwind record, also called a call stack record.

For example, the following process may occur during the profiling phase if the user of the profiling utility has requested sample-based profiling information. Each time a particular type of timer interrupt occurs, a trace record is written, which indicates the system program counter. This system program counter may be used to identify the routine that is interrupted. In the depicted example, a timer interrupt is used to initiate gathering of trace data. Of course, other types of interrupts may be used other than timer interrupts. Interrupts based on a programmed performance monitor event or other types of periodic events may be employed..

In the post-processing phase 504, the data collected in the buffer is sent to a file for post-processing. In one configuration, the file may be sent to a server, which determines the profile for the processes on the client machine. Of course, depending on available resources, the post-processing also may be performed on the client machine. In post-processing phase 504, B-trees and/or hash tables may be employed to maintain

names associated with the records in the trace file to be processed. A hash table employs hashing to convert an identifier or a key, meaningful to a user, into a value for the location of the corresponding data in the table. While processing trace records, the B-trees and/or hash tables are updated to reflect the current state of the client machine, including newly loaded jitted code or unloaded code. Also, in the post-processing phase 504, each trace record is processed in a serial manner. As soon as the indicator is encountered and all of the startup information has been processed, event-based trace records from trace hooks and sample-based trace records from timer interrupts are then processed. Timer interrupt information from the timer interrupt records are resolved with existing hash tables. In addition, this information identifies the thread and function being executed. The data is stored in hash tables with a count identifying the number of timer tick occurrences associated with each way of looking at the data. After all of the trace records are processed, the information is formatted for output in the form of a report.

Alternatively, trace information may be processed on the fly so that trace data structures are maintained during the profiling phase. In other words, while a profiling function, such as a timer interrupt, is executing, rather than (or in addition to) writing trace records to a buffer or file, the trace record information is processed to construct and maintain any appropriate data structures.

For example, during the processing of a timer interrupt during the profiling phase, a determination could be made as to whether the code being interrupted is

being interpreted by the Java interpreter. If the code being interrupted is interpreted, the method ID of the method being interpreted may be placed in the trace record. In addition, the name of the method may be obtained and placed in the appropriate B-tree. Once the profiling phase has completed, the data structures may contain all the information necessary for generating a profile report without the need for post-processing of the trace file.

With reference now to **Figure 6**, a flowchart depicts a process used by a trace program for generating trace records from processes executing on a data processing system. **Figure 6** provides further detail concerning the generation of trace records that were not described with respect to **Figure 5**.

Trace records may be produced by the execution of small pieces of code called "hooks". Hooks may be inserted in various ways into the code executed by processes, including statically (source code) and dynamically (through modification of a loaded executable). This process is employed after trace hooks have already been inserted into the process or processes of interest. The process begins by allocating a buffer (step **600**), such as buffer **404** in **Figure 4**. Next, in the depicted example, trace hooks are turned on (step **602**), and tracing of the processes on the system begins (step **604**). Trace data is received from the processes of interest (step **606**). This type of tracing may be performed during phases **500** and/or **502**. This trace data is stored as trace records in the buffer (step **608**). A determination is made as to whether tracing has finished

(step 610). Tracing finishes when the trace buffer has been filled or the user stops tracing via a command and requests that the buffer contents be sent to file. If tracing has not finished, the process returns to step 606 as described above.

Otherwise, when tracing is finished, the buffer contents are sent to a file for post-processing (step 612). A report is then generated in post-processing (step 614) with the process terminating thereafter.

Although the depicted example uses post-processing to analyze the trace records, the processes of the present invention may be used to process trace information in real-time depending on the implementation.

With reference now to **Figure 7**, a flowchart depicts a process that may be used during an interrupt handler trace hook.

The process begins by obtaining a program counter (step 700). Typically, the program counter is available in one of the saved program stack areas. Thereafter, a determination is made as to whether the code being interrupted is interpreted code (step 702). This determination may be made by determining whether the program counter is within an address range for the interpreter used to interpret bytecodes. If the code being interrupted is interpreted, a method block address is obtained for the code being interpreted. A trace record is then written (step 706). The trace record is written by sending the trace information to a trace program, such as trace program 400, which generates trace records for post-processing in the depicted example. This trace record is referred to as an interrupt record, or an interrupt hook.

This type of trace may be performed during phase 502. Alternatively, a similar process, i.e. determining whether code that was interrupted is interpreted code, may occur during post-processing of a trace file.

In addition to event-based profiling, a set of processes may be employed to obtain sample-based profiling information. As applications execute, the applications may be periodically interrupted in order to obtain information about the current runtime environment. This information may be written to a buffer or file for post-processing, or the information may be processed on the fly into data structures representing an ongoing history of the runtime environment. **Figures 8 and 9** describe sample-based profiling in more detail.

A sample-based profiler obtains information from the stack of an interrupted thread. The thread is interrupted by a timer interrupt presently available in many operating systems. The user of the trace facility selects either the program counter option or the stack unwind option, which may be accomplished by enabling one major code or another major code, as described further below. This timer interrupt is employed to sample information from a call stack. By walking back up the call stack, a complete call stack can be obtained for analysis. A "stack walk" may also be described as a "stack unwind", and the process of "walking the stack" may also be described as "unwinding the stack". Each of these terms illustrates a different metaphor for the process. The process can be described as "walking" as the process must obtain and process the stack frames step-by-step. The process may also be described as "unwinding" as the process must obtain and process the stack frames that point to one another, and these pointers

and their information must be "unwound" through many pointer dereferences.

The stack unwind follows the sequence of functions/method calls at the time of the interrupt. A call stack is an ordered list of routines plus offsets within routines (i.e. modules, functions, methods, etc.) that have been entered during execution of a program. For example, if routine A calls routine B, and then routine B calls routine C, while the processor is executing instructions in routine C, the call stack is ABC. When control returns from routine C back to routine B, the call stack is AB. For more compact presentation and ease of interpretation within a generated report, the names of the routines are presented without any information about offsets. Offsets could be used for more detailed analysis of the execution of a program, however, offsets are not considered further herein.

Thus, during timer interrupt processing or at post-processing, the generated sample-based profile information reflects a sampling of call stacks, not just leaves of the possible call stacks, as in some program counter sampling techniques. A leaf is a node at the end of a branch, i.e. a node that has no descendants. A descendant is a child of a parent node, and a leaf is a node that has no children.

With reference now to **Figure 8**, a diagram depicts the call stack containing stack frames. A "stack" is a region of reserved memory in which a program or programs store status data, such as procedure and function call addresses, passed parameters, and sometimes local variables. A "stack frame" is a portion of a thread's stack that represents local storage (arguments, return

addresses, return values, and local variables) for a single function invocation. Every active thread of execution has a portion of system memory allocated for its stack space. A thread's stack consists of sequences of stack frames. The set of frames on a thread's stack represent the state of execution of that thread at any time. Since stack frames are typically interlinked (e.g., each stack frame points to the previous stack frame), it is often possible to trace back up the sequence of stack frames and develop the "call stack". A call stack represents all not-yet-completed function calls -- in other words, it reflects the function invocation sequence at any point in time.

Call stack **800** includes information identifying the routine that is currently running, the routine that invoked it, and so on all the way up to the main program.

Call stack **800** includes a number of stack frames **802**, **804**, **806**, and **808**. In the depicted example, stack frame **802** is at the top of call stack **800**, while stack frame **808** is located at the bottom of call stack **800**. The top of the call stack is also referred to as the "root". The timer interrupt (found in most operating systems) is modified to obtain the program counter value (pcv) of the interrupted thread, together with the pointer to the currently active stack frame for that thread. In the Intel architecture, this is typically represented by the contents of registers: EIP (program counter) and EBP (pointer to stack frame). By accessing the currently active stack frame, it is possible to take advantage of the (typical) stack frame linkage convention in order to chain all of the frames together. Part of the standard linkage convention also dictates that the function return

DRAFT - 60

address be placed just above the invoked-function's stack frame; this can be used to ascertain the address for the invoked function. While this discussion employs an Intel-based architecture, this example is not a restriction. Most architectures employ linkage conventions that can be similarly navigated by a modified profiling interrupt handler.

When a timer interrupt occurs, the first parameter acquired is the program counter value. The next value is the pointer to the top of the current stack frame for the interrupted thread. In the depicted example, this value would point to EBP **808a** in stack frame **808**. In turn, EBP **808** points to EBP **806a** in stack frame **806**, which in turn points to EBP **804a** in stack frame **804**. In turn, this EBP points to EBP **802a** in stack frame **802**. Within stack frames **802-808** are EIPs **802b-808b**, which identify the calling routine's return address. The routines may be identified from these addresses. Thus, routines are defined by collecting all of the return addresses by walking up or backwards through the stack.

With reference now to the **Figure 9**, an illustration of a call stack is depicted. A call stack, such as call stack **900** is obtained by walking the call stack. A call stack is obtained each time a periodic event, such as, for example, a timer interrupt occurs. These call stacks may be stored as call stack unwind trace records within the trace file for post-processing or may be processed on-the-fly while the program continues to execute. In the depicted example, call stack **900** contains a pid **902**, which is the process identifier, and a tid **904**, which is the thread identifier. Call stack **900** also

contains addresses **addr1 906**, **addr2 908** ... **addrN 910**. In this example, **addr1 906** represents the value of the program counter at the time of the interrupt. This address occurs somewhere within the scope of the interrupted function. **addr2 908** represents an address within the process that called the function that was interrupted. For Intel-processor-based data processing systems, it represents the return address for that call; decrementing that value by 4 results in the address of the actual call, also known as the call-site. This corresponds with **EIP 808b** in **Figure 8**; **addrN 910** is the top of the call stack (**EIP 802b**). The call stack that would be returned if the timer interrupt interrupted the thread whose call stack state is depicted in **Figure 8** would consist of: a pid, which is the process id of the interrupted thread; a tid, which is the thread id for the interrupted thread; a pcv, which is a program counter value (not shown on **Figure 8**) for the interrupted thread; **EIP 808b**; **EIP 806b**; **EIP 804b**; and **EIP 802b**. In terms of **Figure 9**, **pcv = addr1**, **EIP 808b = addr2**, **EIP 806b = addr3**, **EIP 804b = addr4**, **EIP 802b = addr5**.

With reference now to **Figure 10A**, a diagram of a program execution sequence along with the state of the call stack at each function entry/exit point is provided. The illustration shows entries and exits occurring at regular time intervals, but this is only a simplification for the illustration. If each function (A, B, C, and X in the figure) were instrumented with entry/exit event hooks, then complete accounting of the time spent within and below each function would be readily obtained. Note in **Figure 10A** that at time 0, the executing thread is in

routine C. The call stack at time 0 is C. At time 1, routine C calls routine A, and the call stack becomes CA and so on. It should be noted that the call stack in **Figure 10A** is a reconstructed call stack that is generated by processing the event-based trace records in a trace file to follow such events as method entries and method exits.

The accounting technique and data structure is described in more detail further below. Unfortunately, this type of instrumentation can be expensive, can introduce bias, and in some cases, can be hard to apply. Sample-based profiling, by sampling the program's call stack, helps to alleviate the performance bias (and other complications) that entry/exit hooks produce.

Consider **Figure 10B**, in which the same program is executed but is being sampled on a regular basis (in the example, the interrupt occurs at a frequency equivalent to two timestamp values). Each sample includes a snapshot of the interrupted thread's call stack. Not all call stack combinations are seen with this technique (note that routine X does not show up at all in the set of call stack samples in **Figure 10B**). This is an acceptable limitation of sampling. The idea is that with an appropriate sampling rate (e.g., 30-1000 times per second), the call stacks in which most of the time is spent will be identified. Although some call stacks are omitted, it is a minor issue provided these call stacks are combinations for which little time is consumed.

In the event-based traces, there is a fundamental assumption that the traces contain information about routine entries and matching routine exits. Often, entry-exit pairs are nested in the traces because

routines call other routines. Time spent (or memory consumed) between entry into a routine and exit from the same routine is attributed to that routine, but a user of a profiling tool may want to distinguish between time spent directly in a routine and time spent in other routines that it calls.

Figure 10C shows an example of the manner in which time may be expended by two routines: a program's "main" calls routine A at time "t" equal to zero; routine A computes for 1 ms and then calls routine B; routine B computes for 8 ms and then returns to routine A; routine A computes for 1 ms and then returns to "main". From the point of view of "main", routine A took 10 ms to execute, but most of that time was spent executing instructions in routine B and was not spent executing instructions within routine A. This is a useful piece of information for a person attempting to optimize the example program. In addition, if routine B is called from many places in the program, it might be useful to know how much of the time spent in routine B was on behalf of (or when called by) routine A and how much of the time was on behalf of other routines.

A fundamental concept in the output provided by the methods described herein is the call stack. The call stack consists of the routine that is currently running, the routine that invoked it, and so on all the way up to main. A profiler may add a higher, thread level with the pid/tid (the process IDs and thread IDs). In any case, an attempt is made to follow the trace event records, such as method entries and exits, as shown in **Figure 10A**, to reconstruct the structure of the call stack frames while the program was executing at various times during

the trace.

The post-processing of a trace file may result in a report consisting of three kinds of time spent in a routine, such as routine A: (1) base time--the time spent executing code in routine A itself; (2) cumulative time (or "cum time" for short)--the time spent executing in routine A plus all the time spent executing every routine that routine A calls (and all the routines they call, etc.); and (3) wall-clock time or elapsed time. This type of timing information may be obtained from event-based trace records as these records have timestamp information for each record.

A routine's cum time is the sum of all the time spent executing the routine plus the time spent executing any other routine while that routine is below it on the call stack. In the example above in **Figure 10C**, routine A's base time is 2 ms, and its cum time is 10 ms. Routine B's base time is 8 ms, and its cum time is also 8 ms because it does not call any other routines. It should be noted that cum time may not be generated if a call stack tree is being generated on the fly--cum time may only be computed after the fact during the post-processing phase of a profile utility.

For wall-clock or elapsed time, if while routine B was running, the system fielded an interrupt or suspended this thread to run another thread, or if routine B blocked waiting on a lock or I/O, then routine B and all the entries above routine B on the call stack accumulate elapsed time but not base or cum time. Base and cum time are unaffected by interrupts, dispatching, or blocking. Base time only increases while a routine is running, and cum time only increases while the routine or a routine

below it on the call stack is running.

In the example in **Figure 10C**, routine A's elapsed time is the same as its cum time--10 ms. Changing the example slightly, suppose there was a 1 ms interrupt in the middle of B, as shown in **Figure 10D**. Routine A's base and cum time are unchanged at 2 ms and 10 ms, but its elapsed time is now 11 ms.

Although base time, cum time and elapsed time were defined in terms of processor time spent in routines, sample based profiling is useful for attributing consumption of almost any system resource to a set of routines, as described in more detail below with respect to **Figure 11B**. Referring to **Figure 10C** again, if routine A initiated two disk I/O's, and then routine B initiated three more I/O's when called by routine A, routine A's "base I/O's" are two and routine A's "cum I/O's" are five. "Elapsed I/O's" would be all I/O's, including those by other threads and processes, which occurred between entry to routine A and exit from routine A. More general definitions for the accounting concepts during profiling would be the following: base--the amount of the tracked system resource consumed directly by this routine; cum--the amount of the tracked system resource consumed by this routine and all routines below it on the call stack; elapsed--the total amount of the tracked system resource consumed (by any routine) between entry to this routine and exit from the routine.

As noted above, **Figures 10A-10D** describe the process by which a reconstructed call stack may be generated by processing the event-based trace records in a trace file by following such events as method entries and method exits. Hence, although **Figures 11A-14** describe call

stack trees that may be applicable to processing sample-based trace records, the description below for generating or reconstructing call stacks and call stack trees in **Figures 11A-14** is mainly directed to the processing of event-based trace records.

With reference now to **Figure 11A**, a diagram depicts a tree structure generated from trace data. This figure illustrates a call stack tree 1100 in which each node in tree structure 1100 represents a function entry point. Additionally, in each node in tree structure 1100, a number of statistics are recorded. In the depicted example, each node, nodes 1102-1108, contains an address (addr), a base time (BASE), cumulative time (CUM) and parent and children pointers. As noted above, this type of timing information may be obtained from event-based trace records as these records have timestamp information for each record. The address represents a function entry point. The base time represents the amount of time consumed directly by this thread executing this function. The cumulative time is the amount of time consumed by this thread executing this function and all functions below it on the call stack. In the depicted example, pointers are included for each node. One pointer is a parent pointer, a pointer to the node's parent. Each node also contains a pointer to each child of the node.

Those of ordinary skill in the art will appreciate that tree structure 1100 may be implemented in a variety of ways and that many different types of statistics may be maintained at the nodes other than those in the depicted example.

The call stack is developed from looking back at all return addresses. These return addresses will resolve

within the bodies of those functions. This information allows for accounting discrimination between distinct invocations of the same function. In other words, if function X has 2 distinct calls to function A, the time associated with those calls can be accounted for separately. However, most reports would not make this distinction.

With reference now to **Figure 11B**, a call stack tree which reflects call stacks observed during a specific example of system execution will now be described. At each node in the tree, several statistics are recorded. In the example shown in **Figure 11B**, the statistics are time-based statistics. The particular statistics shown include the number of distinct times the call stack is produced, the sum of the time spent in the call stack, the total time spent in the call stack plus the time in those call stacks invoked from this call stack (referred to as cumulative time), and the number of instances of this routine above this instance (indicating depth of recursion).

For example, at node 1152 in **Figure 11B**, the call stack is CAB, and the statistics kept for this node are 2:3:4:1. Note that call stack CAB is first produced at time 2 in **Figure 10A**, and is exited at time 3. Call stack CAB is produced again at time 4, and is exited at time 7. Thus, the first statistic indicates that this particular call stack, CAB, is produced twice in the trace. The second statistic indicates that call stack CAB exists for three units of time (at time 2, time 4, and time 6). The third statistic indicates the cumulative amount of time spent in call stack CAB and those call stacks invoked from call stack CAB (i.e.,

00000000000000000000000000000000

those call stacks having CAB as a prefix, in this case CABB). The cumulative time in the example shown in **Figure 11B** is four units of time. Finally, the recursion depth of call stack CAB is one, as none of the three routines present in the call stack have been recursively entered.

Those skilled in the art will appreciate that the tree structure depicted in **Figure 11B** may be implemented in a variety of ways, and a variety of different types of statistics may be maintained at each node. In the described embodiment, each node in the tree contains data and pointers. The data include the name of the routine at that node, and the four statistics discussed above. Of course, many other types of statistical information may be stored at each node. In the described embodiment, the pointers for each node include a pointer to the node's parent, a pointer to the first child of the node (i.e. the left-most child), a pointer to the next sibling of the node, and a pointer to the next instance of a given routine in the tree. For example, in **Figure 11B**, node **1154** would contain a parent pointer to node **1156**, a first child pointer to node **1158**, a next sibling pointer equal to NULL (note that node **1154** does not have a next sibling), and a next instance pointer to node **1162**.

The type of performance information and statistics maintained at each node are not constrained to time-based

performance statistics. The present invention may be used to present many types of trace information in a compact manner, which supports performance queries. For example, rather than keeping statistics regarding time, tracing may be used to track the number of Java bytecodes executed in each method (i.e., routine) called. The tree structure of the present invention would then contain statistics regarding bytecodes executed rather than time. In particular, the quantities recorded in the second and third categories would reflect the number of bytecodes executed rather than the amount of time spent in each method.

Tracing may also be used to track memory allocation and deallocation. Every time a routine creates an object, a trace record could be generated. The tree structure of the present invention would then be used to efficiently store and retrieve information regarding memory allocation. Each node would represent the number of method calls, the amount of memory allocated within a method, the amount of memory allocated by methods called by the method, and the number of methods above this instance (i.e., the measure of recursion). Those skilled in the art will appreciate that the tree structure of the present invention may be used to represent a variety of performance data in a manner which is very compact, and allows a wide variety of performance queries to be performed.

The tree structure shown in **Figure 11B** depicts one way in which data may be pictorially presented to a user. The same data may also be presented to a user in tabular form as shown in **Figure 12**.

With reference now to **Figure 12**, a call stack tree

DEPARTMENT OF DEFENSE

presented as a table will now be described. Note that **Figure 12** contains a routine, `pt_piddid`, which is the main process/thread, which calls routine C. Table 12 includes columns of data for Level 1230, RL 1232, Calls 1234, Base 1236, Cum 1238, and Indent 1240. Level 1230 is the tree level (counting from the root as level 0) of the node. RL 1232 is the recursion level. Calls 1234 is the number of occurrences of this particular call stack, i.e., the number of times this distinct call stack configuration occurs. Base 1236 is the total observed time in the particular call stack, i.e., the total time that the stack had exactly these routines on the stack. Cum 1238 is the total time in the particular call stack plus deeper levels below it. Indent 1240 depicts the level of the tree in an indented manner. From this type of call stack configuration information, it is possible to infer each unique call stack configuration, how many times the call stack configuration occurred, and how long it persisted on the stack. This type of information also provides the dynamic structure of a program, as it is possible to see which routine called which other routine. However, there is no notion of time-order in the call stack tree. It cannot be inferred that routines at a certain level were called before or after other routines on the same level.

The pictorial view of the call stack tree, as illustrated in **Figure 11B**, may be built dynamically or built statically using a trace text file or binary file as input. **Figure 13** depicts a flow chart of a method for building a call stack tree using a trace text file as input. In **Figure 13**, the call stack tree is built to

illustrate module entry and exit points.

With reference now to **Figure 13**, it is first determined if there are more trace records in the trace text file (step 1350). If so, several pieces of data are obtained from the trace record, including the time, whether the event is an enter or an exit, and the module name (step 1352). Next, the last time increment is attributed to the current node in the tree (step 1354). A check is made to determine if the trace record is an enter or an exit record (step 1356). If it is an exit record, the tree is traversed to the parent (using the parent pointer), and the current tree node is set equal to the parent node (step 1358). If the trace record is an enter record, a check is made to determine if the module is already a child node of the current tree node (step 1360). If not, a new node is created for the module and it is attached to the tree below the current tree node (step 1362). The tree is then traversed to the module's node, and the current tree node is set equal to the module node (step 1364). The number of calls to the current tree node is then incremented (step 1366). This process is repeated for each trace record in the trace output file, until there are no more trace records to parse (step 1368).

With reference now to **Figure 14**, a flow chart depicts a method for building a call stack tree dynamically as tracing is taking place during system execution. In **Figure 14**, as an event is logged, it is added to the tree in real time. Preferably, a call stack tree is maintained for each thread. The call stack tree reflects the call stacks recorded to date, and a current

tree node field indicates the current location in a particular tree. When an event occurs (step 1470), the thread ID is obtained (step 1471). The time, type of event (i.e., in this case, whether the event is a method entry or exit), the name of the module (i.e., method), location of the thread's call stack, and location of the thread's "current tree node" are then obtained (step 1472). The last time increment is attributed to the current tree node (step 1474). A check is made to determine if the trace event is an enter or an exit event (step 1476). If it is an exit event, the tree is traversed to the parent (using the parent pointer), and the current tree node is set equal to the parent node (step 1478). At this point, the tree can be dynamically pruned in order to reduce the amount of memory dedicated to its maintenance (step 1479). Pruning is discussed in more detail below. If the trace event is an enter event, a check is made to determine if the module is already a child node of the current tree node (step 1480). If not, a new node is created for the module, and it is attached to the tree below the current tree node (step 1482). The tree is then traversed to the module's node, and the current tree node is set equal to the module node (step 1484). The number of calls to the current tree node is then incremented (step 1486). Control is then passed back to the executing module, and the dynamic tracing/reduction program waits for the next event to occur (step 1488).

One of the advantages of using the dynamic tracing/reduction technique described in **Figure 14** is its enablement of long-term system trace collection with a

finite memory buffer. Very detailed performance profiles may be obtained without the expense of an "infinite" trace buffer. Coupled with dynamic pruning, the method depicted in **Figure 14** can support a fixed-buffer-size trace mechanism.

The use of dynamic tracing and reduction (and dynamic pruning in some cases) is especially useful in profiling the performance characteristics of long running programs. In the case of long running programs, a finite trace buffer can severely impact the amount of useful trace information that may be collected and analyzed. By using dynamic tracing and reduction (and perhaps dynamic pruning), an accurate and informative performance profile may be obtained for a long running program.

Many long-running applications reach a type of steady-state, where every possible routine and call stack is present in the tree and updating statistics. Thus, trace data can be recorded and stored for such applications indefinitely within the constraints of a bounded memory requirement using dynamic pruning. Pruning has value in reducing the memory requirement for those situations in which the call stacks are actually unbounded. For example, unbounded call stacks are produced by applications that load and run other applications.

Pruning can be performed in many ways, and a variety of pruning criteria is possible. For example, pruning decisions may be based on the amount of cumulative time attributed to a subtree. Note that pruning may be disabled unless the amount of memory dedicated to maintaining the call stack exceeds some limit. As an exit event is encountered (such as step **1478** in Figure

14), the cumulative time associated with the current node is compared with the cumulative time associated with the parent node. If the ratio of these two cumulative times does not exceed a pruning threshold (e.g., 0.1), then the current node and all of its descendants are removed from the tree. The algorithm to build the tree proceeds as before by traversing to the parent, and changing the current node to the parent.

Many variations of the above pruning mechanism are possible. For example, the pruning threshold can be raised or lowered to regulate the level of pruning from very aggressive to none. More global techniques are also possible, including a periodic sweep of the entire call stack tree, removing all subtrees whose individual cumulative times are not a significant fraction of their parent node's cumulative times.

Data reduction allows analysis programs to easily and quickly answer many questions regarding how computing time was spent within the traced program. This information may be gathered by "walking the tree" and accumulating the data stored at various nodes within the call stack tree, from which it can be determined the amount of time spent strictly within routine A, the total amount of time spent in routine A and in the routines called by routine A either directly or indirectly, etc.

With reference now to **Figure 15A**, a flowchart depicts a process for creating a call stack tree structure from call stack unwind records in a trace file.

Figures 10A-14 above primarily showed the processes involved in generating a call stack tree from event-based trace records, which show events such as method entries and method exits. These types of trace records allow a

call stack to be generated, usually during a postprocessing phase of the profile tool or utility. Using timer interrupts, a profiling function may walk an active call stack to generate a call stack unwind trace record. **Figure 15A** describes a process for combining the information in a call stack unwind trace record into a call stack tree. The call stack tree may have been previously constructed from other call stack unwind trace records or from event-based trace records according to the methods described in **Figures 10A-14**.

The process begins by reading a call stack unwind record (step 1500). This step processes the call stack information in the record to determine what routines are or were executing when the timer interrupt occurs or occurred, depending on whether the call stack unwind record is being processed on-the-fly or is being postprocessed. A sample-based profiling function avoids, through the call stack unwind, the need for adding additional instructions to the program, which affects the performance and time spent in routines. Next, the tree structure for this process/thread (pid, tid) is located (step 1502). Then, the pointer (PTR) is set to the root of this tree structure by setting PTR = root(pid, tid) (step 1504). The index is set equal to N, which is the number of entries in the call stack (step 1506).

A determination is made as to whether the index is equal to zero (step 1508). If the index is equal to zero, the process then returns to determine whether additional call stack unwind trace records are present for processing (step 1510). If additional call stack unwind trace records are present, the process then returns to step 1500 to read another call stack unwind

0000000000000000

trace record. Otherwise, the process terminates.

On the other hand, if the index is not equal to zero, the process then sets sample_address equal to the call_stack_address[index] (step 1512). The B-tree is then used to lookup the address to get a routine name (step 1513). Next, a determination is made as to whether PTR.child.name for any child of PTR is equal to the looked-up routine name (step 1514). In other words, this step determines whether the routine name has ever been seen at this level in the tree structure. If the address has never been seen at this level in the tree structure, a new child of PTR is created and the PTR.child.name is set equal to the routine name, the variable PTR.child.BASE for the node is set equal to zero, and the variable PTR.child.CUM for the node is set equal to zero (step 1516). Thereafter, the cumulative time for the node is incremented by incrementing the variable PTR.child.CUM (step 1518). The process also proceeds to step 1518 from step 1514 if the address has been seen at this level. In the case of sample-based trace records, the "cumulative" time represents the number of times that this particular call stack configuration has been processed.

Next, a determination is made as to whether the sample address, sample_address, is equal the last address in the call stack sample, call_stack_address[1] (step 1520). If the sample address is equal to the address being processed, the base time for the node is incremented by incrementing the variable PTR.child.BASE (step 1522). The pointer PTR is then set equal to the child (step 1524), and the index is decremented (step

1526) with the process then returning to step 1508 as previously described. With reference again to step 1520, if the sample address is not equal to the address being processed, the process then proceeds to step 1524.

In the depicted example in **Figure 15A**, the process is used to process call stack unwind records recorded during execution of a program. The illustrated process also may be implemented to dynamically process call stack unwind records during execution of a program. For example, step 1510 may be modified to wait until the next timer interrupt occurs and then continue to loop back to step 1510 at the next interrupt.

The addresses obtained during sampling are used to identify functions. The functions are identified by mapping these addresses into functions.

With reference now to **Figure 15B**, a flowchart depicts a process for identifying functions from an address obtained during sampling. The process begins by reading a program counter value that is obtained during sampling of the call stack (step 1550). A determination is made as to whether the end of file has been reached (step 1552). If the end of the file has not been reached, the program counter value is looked up in a global map (step 1554). A global map in the depicted example is a map of system and per process symbols that are generated from system loader information and application, library, and system symbol tables. A process plus function id is obtained from the global map in response to looking up the program counter value (step 1556). Thereafter, the process returns to step 1550. The function information may be used in generating

Docket No. AUS990853US1

reports, such as those described below. The process in **Figure 15B** also may be used during execution of a program that is sampled.

With reference now to **Figure 16**, a diagram of a record generated using the processes of present invention is depicted. Each routine in record 1600 is listed separately, along with information regarding the routine in **Figure 16**. For example, Calls column 1604 lists the number of times each routine has been called. BASE column 1606 contains the total time spent in the routine, while CUM column 1608 includes the cumulative time spent in the routine and all routines called by the routine. Name column 1612 contains the name of the routine. With reference now to **Figure 17**, a diagram of another type of report that may be produced is depicted. The report depicted in **Figure 17** illustrates much of the same information found in **Figure 16**, but in a slightly different format. As with **Figure 16**, diagram 1700 includes information on calls, base time, and cumulative time.

Figure 17 shows a sample-based trace output containing times spent within various routines as measured in microseconds. **Figure 17** contains one stanza (delimited by horizontal lines) for each routine that appears in the sample-based trace output. The stanza contains information about the routine itself on the line labeled "Self", about who called it on lines labeled "Parent", and about who the routine called on lines labeled "Child". The stanzas are in order of cum time. The third stanza is about routine A, as indicated by the line beginning with "Self." The numbers on the "Self"

line of this stanza show that routine A was called three times in this trace, once by routine C and twice by routine B. In the profile terminology, routines C and B are (immediate) parents of routine A. Routine A is a child of routines C and B. All the numbers on the "Parent" rows of the second stanza are breakdowns of routine A's corresponding numbers. Three microseconds of the seven microsecond total base time spent in A was when it was called by routine C, and three microseconds when it was first called by routine B, and another one microsecond when it was called by routine B for a second time. Likewise, in this example, half of routine A's fourteen microsecond cum time was spent on behalf of each parent.

Looking now at the second stanza, we see that routine C called routine B and routine A once each. All the numbers on "Child" rows are subsets of numbers from the child's profile. For example, of the three calls to routine A in this trace, one was by routine C; of routine A's seven microsecond total base time, three microseconds were while it was called directly by routine C; of routine A's fourteen microsecond cum time, seven microseconds was on behalf of routine C. Notice that these same numbers are the first row of the third stanza, where routine C is listed as one of routine A's parents.

The four relationships that are true of each stanza are summarized at the top of **Figure 17**. First, the sum of the numbers in the Calls column for parents equals the number of calls on the self row. Second, the sum of the numbers in the Base column for parents equals Self's base. Third, the sum of the numbers in the Cum column for parents equals Self's Cum. These first three

invariants are true because these characteristics are the definition of Parent; collectively they are supposed to account for all of Self's activities. Fourth, the Cum in the Child rows accounts for all of Self's Cum except for its own Base.

Program sampling contains information from the call stack and provides a profile, reflecting the sampling of an entire call stack, not just the leaves. Furthermore, the sample-based profiling technique may also be applied to other types of stacks. For example, with Java programs, a large amount of time is spent in a routine called the "interpreter". If only the call stack was examined, the profile would not reveal much useful information. Since the interpreter also tracks information in its own stack, e.g., a Java stack (with its own linkage conventions), the process can be used to walk up the Java stack to obtain the calling sequence from the perspective of the interpreted Java program.

With reference now to **Figure 18**, a figure depicts a report generated from a trace file containing both event-based profiling information (method entry/exits) and sample-based profiling information (stack unwinds). **Figure 18** is similar to **Figure 12**, in which a call stack tree is presented as a report, except that **Figure 18** contains embedded stack walking information. Call stack tree **1800** contains two stack unwinds generated within the time period represented by the total of 342 ticks. Stack unwind identifier **1802** denotes the beginning of stack unwind information **1806**, with the names of routines that are indented to the right containing the stack information that the stack walking process was able to discern. Stack unwind identifier **1804** denotes the

P00000000000000000000000000000000

beginning of stack unwind information **1808**. In this example, "J:" identifies an interpreted Java method and "F:" identifies a native function, such as a native function within JavaOS. A call from a Java method to a native method is via "ExecuteJava." Hence, at the point at which the stack walking process reaches a stack frame for an "ExecuteJava," it cannot proceed any further up the stack as the stack frames are discontinued. The process for creating a tree containing both event-based nodes and sample-based nodes is described in more detail further below. In this case, identifiers **1802** and **1804** also denote the major code associated with the stack unwind.

With reference now to **Figure 19**, a table depicts major codes and minor codes that may be employed to instrument software modules for profiling. In order to facilitate the merging of event-based profiling information and sample-based profiling information, a set of codes may be used to turn on and off various types of profiling functions.

For example, as shown in **Figure 19**, the minor code for a stack unwind is designated as **0x7fffffff**, which may be used for two different purposes. The first purpose, denoted with a major code of **0x40**, is for a stack unwind during a timer interrupt. When this information is output into a trace file, the stack information that appears within the file will have been coded so that the stack information is analyzed as sample-based profiling information. The second purpose, denoted with a major code of **0x41**, is for a stack unwind in an instrumented routine. This stack information could then be post-processed as event-based profiling information.

Other examples in the table show a profile or major code purpose of tracing jitted methods with a major code value of 0x50. Tracing of jitted methods may be distinguished based on the minor code that indicates method invocation or method exit. In contrast, a major code of 0x30 indicates a profiling purpose of instrumenting interpreted methods, while the minor code again indicates, with the same values, method invocation or method exit.

Referring back to **Figure 18**, the connection can be made between the use of major and minor codes, the instrumentation of code, and the post-processing of profile information. In the generated report shown in **Figure 18**, the stack unwind identifiers can be seen to be equal to 0x40, which, according to the table in **Figure 19**, is a stack unwind generated in response to a timer interrupt. This type of stack unwind may have occurred in response to a regular interrupt that was created in order to generate a sampled profile of the executing software.

As noted in the last column of the table in **Figure 19**, by using a utility that places a hook into a software module to be profiled, a stack unwind may be instrumented into a routine. If so, the output for this type of stack unwind will be designated with a major code of 0x41.

When gathering profile information using an symmetric multiprocessing (SMP) system, if two processors update the same metric variable concurrently, it is impossible to apportion the change in a metric variable between two processors. If two processors update the same global variable concurrently, it cannot be determined how much of the change is due to a thread on a

first processor and how much of the change is due to a second thread on a second processor.

Profile information is gathered about the execution of a particular program or routine by accumulating changes in specific metric values and analyzing the execution performance from those accumulated metric values. In order for a profiler to apply the changes to a metric, such as, bytes allocated to a method, the Jvm must either call the profiler whenever a metric event occurs or an interface must be to allow the profiler to determine the value of the change in the metric for the current thread. Two methodologies are identified below to provide this support in an SMP system. **Figures 20 - 24** depict a first approach wherein the profiler to keep track of the changes itself on a per thread basis as the changes are being made. **Figures 25 - 31** depict a second approach in which individual processors maintain per processor data areas. There, the kernel or a kernel device driver updates metric values for a thread whenever a thread switch occurs or there is a request for the metric on a specified thread.

With reference now to **Figure 20**, a block diagram depicts the relationship between a profiler and other software components in a data processing system capable of application level thread oriented process for tracking metrics on a per thread basis. The process of the present embodiment tracks metrics equally well regardless whether processing is provided by a uniprocessor or SMP architecture.

Operating system kernel **2000** provides native support for the execution of programs and applications, such as Jvm **2002**, in a data processing system. Jvm **2002** executes

Java programs, possibly by compiling the program via a just in time (JIT) compiler (not shown). As Java applications execute, objects are allocated and deallocated in a heap, and Jvm 2002 maintains heap information 2006 concerning the objects, such as heap 1960 shown in **Figure 19**. Profiler 2008 accepts events from Jvm 2002 instrumentation through Jvm Profiling Interface (JVMPPI) 2010, and returns information as required by Jvm. Preferably, profiler 2008 is a set of native runtime DLLs (dynamic link libraries) supported by operating system kernel 2000. Profiler 2008 generates call stack trees, trace output 2012, etc. as necessary to provide a runtime profile to an application developer monitoring the execution of a profiled program.

In a preferred embodiment profiler 2008 allocates memory space in its own buffer for metrics (M) being tracked. Here, the notation 'M' represents values for individual metrics $m_1, m_2, m_3 \dots m_j$. It should be understood that reference to metrics (M) refers to any one or all of individual metrics $m_1, m_2, m_3 \dots m_j$, where $M = m_1, m_2, m_3 \dots m_j$. An individual metric (m) is any metric normally tracked for monitoring method execution, performance optimization or memory leak detection, such as number of allocated objects or bytes to a method or execution time. Metrics values may be tracked as variables, for example the accumulated value of metrics (M) is represented as Cum (M), while the accumulated value of metrics (M) for thread (t) is represented as variable Cum (M,t) and the last accumulated value of metrics (M) accumulated in response to the last metric changing event for thread (t) is represented as LastCum (M,t).

Profiler 2008 may use a variety of data structures for maintaining the metric values, such as, a linked list. As a new thread is launched, a new element may be added to the linked list for storing metric values for that thread. Alternatively, the profiler has its own buffer area or "mini-heap" from which it may allocate space for each metric data area. Profiler 2008 allocates data areas in its buffer and stores values for: a global variable representing the total value of all metrics (M) for all threads, Total (M) 2007; variables representing the accumulated values of all metrics (M) for all active threads ($t_0 - t_k$), Cum (M, t_0) 2030 to Cum (M, t_k) 2034; and variables representing the last accumulated values of all metrics (M) for all active threads ($t_0 - t_k$) at the time of the last Jvm event, LastCum (M, t_0) 2031 to LastCum (M, t_k) 2035.

Jvm 2002 utilizes operating system kernel 2000 for processing support. Operating system 2000 controls one or more processes for processing threads for Jvm 2002. In a uniprocessor environment, the profiler captures variable values for a metric at thread switches and at entries and exits and applies the change to the method on the specified thread. However, in the depicted example, operating system kernel 2000 supports symmetric multiprocessing (SMP) which allows multiple processors to operate simultaneously. In this case, processors P_0 2020 - P_n 2028 are supported by operating system kernel 2000 in a multiprocessing fashion. In the SMP case, if two processors update the same global variable concurrently, it is impossible to apportion the change in a metric to the correct thread without somehow SMP

enabling the profiler. While a non-SMP enabled profiler would capture metric values for an individual processor at thread switches and at entries and exits for a method on a specified thread, individual values from each SMP processor are impossible to determine.

In accordance with a preferred embodiment of the present invention, profiler 2008 accurately tracks metric values regardless of the type of processor architecture. For example, operating system kernel 2000 may utilize a single processor (uniprocessor architecture) or may instead utilize multiple processors in a symmetrical multiprocessor (SMP) architecture as depicted in the present figure by processors P₀ 2020 to P_n 2028. Each time a thread is launched, Jvm 2002 creates a new Java stack for the thread. The thread's Java stack stores the state of the Java method invocations for the thread. Jvm 2002 also calls operating system kernel 2000 for processor support for the new thread. Operating system kernel 2000 dispatches the thread to any one of processors P₀ 2020 to P_n 2028 which perform the necessary thread processing completely hidden from Jvm 2002 and profiler 2008. However, profiler 2008 need not receive any metric information from operating system kernel 2000 because Jvm 2002 communicates events which result in a change in any metric value directly to profiler 2008 through JVMPPI 2010. Thus, profiler 2008 can calculate the value of any metric being tracked without communicating with operating system kernel 2000. Jvm 2002 merely sends metric events and method entry and exit events to profiler 2008 when calling operating system kernel 2000 for processor support. Profiler 2008 can

0
9
8
7
6
5
4
3
2
1
0

then update the metric values for the current thread as well for the current method based on information provided by Jvm 2002. On the occasion when Jvm 2002 needs metric values for a specific variable and/or thread, the Jvm merely requests the needed metric variable from profiler 2008 through JVMPi 2010.

With reference to **Figures 21A** and **21B**, flowcharts depict the initialization process necessary for performing an application level thread-oriented approach for tracking metrics in accordance with a preferred embodiment of the present invention. **Figure 21A** is a flowchart illustrating a process for initializing the profiler for performing the application level thread-oriented approach for tracking metrics (M) in accordance with a preferred embodiment of the present invention. The process begins with the Jvm being called by a method for processing current thread (t) (step 2102). Prior to current thread (t) being launched the profiler had not been tracking any metric variables. Therefore, the profiler must be initialized for tracking specific metrics. The Jvm then initializes the profiler, via JVMPi by sending a list of the specific metrics for the profiler to track (step 2104). These metrics might include, for example, object and byte allocations for a method, active objects, live bytes or execution time. The profiler then allocates data areas within its buffer for tracking the metric variables total (M), Cum (M,t) and LastCum (M,t) on current thread (t) (step 2106). The process for initializing the profiler for metrics (M) on current thread (t) is then complete.

Figure 21B is a flowchart illustrating the process for allocating new data areas in response to a receiving

an entry or exit thread event on new thread (t_{new}) for the previously initialized metrics (M). Each time a thread event occurs, profiler is responsible for tracking metrics (M) on that thread. If space has not been previously allocated for new thread (t_{new}), space must be allocated by the profiler before metrics can be tracked for that thread. Each time a thread event occurs on a new thread, the profiler must allocate data areas for the metric variables Cum (M, t_{new}) and LastCum (m, t_{new}) (step 2110). With the data areas allocated for each new thread (t_{new}), the process is complete.

Once the data area has been allocated for metrics (M), the profiler is enabled to track metrics saved in the profiler's data areas. Metric values are updated in two basic ways, first in response to a method entry or exit event. There all metrics (M) are updated with respect to the current method on the basis of a thread event occurring on the current thread (t). The second is in response to a single metric (m) event. There, an individual metric (m) is updated on the basis of the an individual metric event occurring.

Referring to **Figure 22**, a flowchart depicting a process for tracking an individual metric in response to a metric event in accordance with a preferred embodiment of the present invention. The process begins with the Jvm having a metric event (step 2202). In this case, the metric event refers to an event associated with any single metric (m), such as number of objects allocated or deallocated, number of bytes allocated or deallocated, time, et cetera. Of course a metric event may involve a plurality of metrics (M) being tracked, in that case the process is duplicated for each metric (m). The process

continues with the Jvm sending a notification of the metric event to the profiler including the change in the value of metric (m), MDelta (m) (step 2204). The profiler receives the metric event and the change in the metric's value, (MDelta (m)) (step 2206). The profiler applies the metric value, MDelta (m), to the appropriate Total and Cum metric values (step 2208). For example, Total (m) is increased by the amount MDelta (m).

Total (m) += MDelta (m)

Additionally, the Cum value of the metric (m) on each thread (t) is increased by MDelta (m).

Cum (m,t) += MDelta (m)

With the Total and Cum values for metrics (M) being updated for the metric event, the process is complete.

With reference to **Figure 23**, a flowchart depicting a process for tracking metrics in response to a method entry or exit event occurring on current thread (t) in accordance with a preferred embodiment of the present invention. The process begins with the profiler receiving an entry or exit event on currently running thread (t) from the Jvm through JVMPPI (step 2302). Here it is assumed that more than a single metric (m) will change as result of the event, however, depending on the individual metrics selected for tracking, it may be the case that only an individual metric (m) changes its value. As the process continues, the profiler must calculate a change in the value of metrics (M) being tracked, Delta (M), since the last method entry or exit event. Delta (M) is the portion of Cum (M,t) which has accumulated since the last method entry or exit event. Finding the value of Delta (M) is possible because the value of LastCum (M,t) was set at the last entry or exit

event. Thus Delta (M) is the difference of Cum (M,t) and LastCum (M,t). Delta (M) can be determined from:

$$\text{Delta (M)} = \text{Cum (M,t)} - \text{LastCum (M,t)}$$

for current thread (t) (step 2304). The profiler then must reset LastCum (M,t) to Cum (M,t) in order that Delta (M) can be calculated for subsequent method entry and exit events (step 2306). LastCum (M,t) is reset by:

$$\text{LastCum (M,t)} = \text{Cum (M,t)}$$

Next, the accumulated metric value for the current method, Method (M), must be updated by the value of the change in metrics (M), Delta (M) (step 2308). The method running on thread (t) is updated by the value of Delta (M) by:

$$\text{Method (M,t)} += \text{Delta (M)}$$

The process for updating a metrics (M) in response to a method entry or exit event is complete.

From time-to-time, the Jvm may need the current value of metrics (M). However, Jvm does not track the metrics themselves and must rely on the profiler for the metric values. Thus, when the Jvm has need for a metric, the Jvm must get that value from the profiler. With reference to **Figure 24**, a flowchart depicting a simplified application programming interface (API) is depicted for retrieving metric values in accordance with a preferred embodiment of the present invention. The process begins with the Jvm requesting metric variable values from the profiler (step 2402). The metric variable may be either the accumulated value of the metrics, Cum (M,t) or the total value of the metrics, Total (M) is the total value of the metrics being tracked for all active threads ($t_0 - t_k$). The profiler then returns requested value(s) to the Jvm (step 2404). The

process is now complete with respect to a simplified application programming interface (API) for retrieving metric values.

The above-described embodiment has an advantage of accurately tracking metrics for either uniprocessor or SMP architectures at the application level. Little or no modifications are needed to the operating system kernel as the profiler does the work. Each time an event occurs, the Jvm communicates the event directly to the profiler, and thus operating system kernel support is not needed. An alternative to the above-described embodiment is having the operating system kernel calculate and store the metric values for active threads. the operating system kernel then returns the metric values to the profiler in response to a request from the profiler. The operating system kernel must be modified to provide metric tracing support to the profiler.

With respect to **Figure 25**, a diagram illustrating the relationship of the Jvm, profiler and an operating system kernel for tracking metrics using an operating system kernel level per processor based approach in accordance with a preferred embodiment of the present invention. Jvm **2502** utilizes operating system kernel **2500** for processing support as described above of processing threads launched by Jvm **2002**. Operating system **2500** controls one or more processors for that function. In contrast with the above-described embodiment, here profiler **2508** relies on operating system kernel **2500** to calculate the accumulated values of metrics (M). Thus, operating system kernel **2500** must allocate memory space for tracking metrics (M).

However, because operating system kernel **2500**

utilizes processors P_0 **2520** through P_n **2528** in a symmetrical multiprocessor (SMP) fashion, changes in metrics (M) must be computed and stored on a per processor basis and then applied to a per thread metric. By first computing the changes in a metric due to a processor (p) for the thread running on processor (p), each SMP processor's contribution can be accurately reflected in the value of the metric for a thread. Thus, operating system kernel **2500** stores the per processor accumulated values of metrics (M) in a data area associated with each processor, $PCum(M, p_0)$ **2530** through $PCum(M, p_n)$ **2538**. Additionally, operating system kernel **2500** stores the value of the last accumulated metric variable for each processor, $LastCum(M, p_0)$ **2540** through $LastCum(M, p_n)$ **2548**. LastCum values are reset for each processor every time $PCum(M, p)$ is used to update the value of a per thread accumulated metric variable. This may occur when a thread is dispatched to processor (p), at method entry or exit events while a thread is being processed on processor (p) or any time the profiler calls for a current value of a metric variable for a thread running on processor (p). $PLastCum(M, p_0)$ **2540** through $PLastCum(M, p_n)$ **2548** are needed for computing the change in the value of metrics (M), ΔM on a processor. Additionally, operating system kernel **2500** maintains a data area for value of per thread accumulated metric variables, $TCum(M, t)$, for active threads (t_0-t_k). These variables are depicted in the diagram as $TCum(M, t_0)$ **2560** through $TCum(M, t_k)$ **2568**.

Thus, in accordance with a preferred embodiment of the present invention, profiler **2508** is relieved of the

DOCKET NO. AUS990853US1

responsibility for computing the accumulated values of metrics (M) for any thread. Instead profiler **2508** merely calls operating system **2500** for the current value of per thread accumulated metric variables, TCum (M,t), for any active thread (t_0-t_k). Therefore, it is only necessary for profiler **2508** to provide data areas for storing the accumulated values of the metrics variables for active threads (t_0-t_k), ProfCum (M,t₀) **2570** through ProfCum (M,t_k) **2574**.

With reference to **Figure 26A**, a flowchart depicting the initialization process for tracking metrics on a per processor basis in accordance with a preferred embodiment of the present invention. The process begins with the Jvm being called by a method for processing current thread (t) (step **2602**). Prior to current thread (t) being launched neither the profiler or the operating system kernel had been tracking any metric variables. It is assumed that no data areas had been allocated for any metric variable in either the profiler or the operating system kernel. The Jvm then passes parameters to the profiler, including the identity of current thread (t) (step **2604**). The profiler then calls the Jvm to indicate identity of metrics variables being tracked (step **2606**). The profiler then allocates a data area for the accumulated value of metrics (M) for current thread (t), ProfCum (m,t) and initializes the metric variable ProfCum (M,t) to zero (step **2608**). ProfCum (M,t) is initialized by:

$$\text{ProfCum (M,t)} = 0$$

Next, the operating system kernel allocates and initializes data areas for metric variables to be tracked

PENDING PUBLICATION

(step 2610). The operating system kernel allocates data areas needed for tracking metrics (M), if necessary, and initializes the value of each data area to zero for each processor and the current thread (t). One of ordinary skill in the art would be familiar with operating systems in which per processor data areas are predefined. In those environments the operating system has a pointer to each per processor data area. The per processor data area pointers can be used by the Jvm, for example, to directly access the per processor data areas and update metric variables contained within those data areas.

The operating system kernel sets to zero the value of the per processor accumulated metric variables, $Pcum(M,p)$, for each processor. The metric variables are stored in each processor's data area. $PCum(M,p)$ is set by:

$$PCum(M,p) = 0$$

Next, the operating system kernel sets the value of the per processor last accumulated metric variables, $PLastCum(M,p)$ to zero for each processor. These metric variables are also stored in each processor's data area. $PLastCum(M,p)$ is set by:

$$PLastCum(M,p) = 0$$

Additionally, operating system kernel sets the accumulated value of the metric for the current thread (t) to zero.

$$TCum(M,t) = 0$$

The process is now complete for the initialization process for tracking metrics on a per processor basis.

With reference to **Figure 26B**, a flowchart depicting a process for initializing the operating system kernel in response to the kernel recognizing a new thread in

accordance with a preferred embodiment of the present invention. Each time a new thread is launched, a data area must be allocated for the metric variables associated with that thread. Of course, the per processor data areas for the metric variables have already been allocated. Additionally, since the new launched thread has not previously been processed, its metric values are nulled (step 2620). Thus, the operating system kernel initializes the per processor accumulated value of the metric variable, PCum (M,p), by setting PCum(M,p) to zero for the processor that will process thread (t), processor (p). The metric variable is stored in processor (p)'s data area. PCum (M,p) is set by:

$$\text{PCum } (M,p) = 0$$

Next, the operating system kernel sets the value of the per processor last accumulated metric variable, PLastCum (M,p) to zero for processor (p). This metric variable are also stored in processor (p)'s data area. PLastCum (M,p) is set by:

$$\text{PLastCum } (M,p) = 0$$

Additionally, operating system kernel allocates a new data area for an accumulated metric variable, TCum (M,t) for the newly launched thread (t). The operating system kernel then initializes that variable by setting it to zero. TCum (M,t) is set by:

$$\text{TCum } (M,t) = 0$$

The process for initializing the operating system kernel in response to the kernel recognizing a new thread is complete.

However, whenever a new thread is launched the profiler must also recognize the new thread and create a

data area for storing that thread's metrics. With reference to **Figure 26C**, a flowchart depicting a process for initializing the profiler in response to the profiler recognizing a new thread in accordance with a preferred embodiment of the present invention. The profiler maintains its own accumulated metric variable values, ProfCum (M,t), for each active thread. Therefore, each time a new thread is launched, the profiler must allocate a data area for the metric variables associated with that thread. Additionally, since the new launched thread has not previously been processed, its metric values are nulled (step 2630). The profiler initializes the per thread accumulated value of the metric variable, ProfCum (M,t), by setting ProfCum(M,t) to zero for the newly launched thread (t). The metric variable is stored in the profiler's data area. ProfCum (M,t) is set by:

$$\text{ProfCum (M,t)} = 0$$

The process for initializing the profiler in response to the profiler recognizing a new thread is complete.

Metric variable values must maintain an accurate count of the metrics being tracked. The metric variable values must be responsive to a variety of events in order to accurately reflect the metric. Because the present embodiment employs an operating system kernel level per processor oriented approach for tracking metric variables, changes in a metric being tracked must be applied at the processor level. The per processor metric variable values can be then used to update the thread level metric variable values.

A change in value of each per processor accumulated metric variable, PCum (M,p), is in response to a variety

of events related to the metrics being tracked. Many of these events are recognized by the processor that is processing the thread. For example, the real time clock or performance counters are updated by the operating system kernel. The change in the metric, M_{Delta} (m) is then applied to PCum (M,p) by that processor. However, other events are not recognized by the processor running the thread and therefore the processor cannot apply M_{Delta} (m) to PCum (M,p) due to those events. In that case, for example, the Jvm must update the metric (m) being tracked to reflect the change in the metric, M_{Delta} (m). One event is when a change in the value of a metric, M_{Delta} (m), occurs at the Jvm. The Jvm is then responsible for updating PCum (m,p) to reflect M_{Delta} (m).

With reference to **Figure 27**, a flowchart depicting a process for the operating system kernel updating a per processor metric variable being tracked, in response to a Jvm request in accordance with a preferred embodiment of the present invention. The process begins with the Jvm recognizing that the value of a metric being tracked, metric (m), has changed by M_{Delta} (m) (**step 2702**). In this case, the metric being tracked is an individual metric (m). The Jvm tracks the Total (m) which is the total accrued value of metrics (m). Therefore, the Jvm must update Total (m) each time the Jvm senses that the value of metric (m) has changed by M_{Delta} (m), whenever M_{Delta} (m) ≠ 0. The metric variable Total (m) is updated by:

Total (m) += M_{Delta} (m)

The Jvm then updates the value of the per processor accumulated metric variable, PCum (m,p) by adding the

value of the change in the metric variable, MDelta (m) (step 2704). The metric variable PCum (m,p) is updated by:

PCum (m,p) += MDelta (m)

While the above-described process is in progress, a thread dispatch may occur. The operating system kernel stops executing the current thread on a first processor and eventually executes the thread on a second processor. In response to a thread dispatch during the process, the value of MDelta (m) may be applied to either the first or second processor's data area as the operating system kernel handles the thread dispatch in accordance with a preferred embodiment described below with respect to **Figure 30**. Updating base metric values with processor accumulated metric values requested from the kernel may be accomplished by the profiler in accordance with a preferred embodiment described below with respect to **Figure 31**. The methods described below for thread dispatch handling and updating base metric values assure that the MDelta (m) is applied as appropriately during a thread dispatch. The process is now complete with respect to the Jvm changing a per processor metric value.

Depending on the type of operating system, the process for updating the value of processor level accumulated metric variable, PCum (m), stored in a processor's data area, may be performed by using one of two updating processes. Indirectly, by the Jvm relying on the operating system kernel to update PCum (m) or directly by the Jvm accessing the processor's data area for the processor that is processing current thread (t) and directly updating the metric variable PCum (m).

With reference to **Figure 28**, a flowchart depicting a

RECORDED - INDEXED - SERIALIZED - FILED

P
A
T
E
N
T
I
M
E
S
O
C
I
E
T
O
R
U
S
A

process for the Jvm directly updating the value of a per processor metric variable in accordance with a preferred embodiment of the present invention. As mentioned above, certain operating systems utilize predefined processor data areas that may be used for storing per processor metrics. These operating systems provide the Jvm with a data area pointer with which to access per processor data areas. With the data area pointer stored in the Jvm's register, the Jvm can directly access a per processor data area and update a per processor metric in a single atomic instruction.

The process begins with the Jvm recognizing that a metric (m) being tracked has changed an amount represented by MDelta (m) (step 2802). Next, the Jvm accesses the data area of the processor processing the current thread (t) and directly updates the accumulated value of a metric, PCum (m,p) stored in the processor's data areas (step 2804). The Jvm utilizes the data area pointer in its register for accessing the processor's data area. One of ordinary skill in the art would realize that updating a per processor variable must be performed as a single atomic instruction before releasing the processor. The metric variable PCum (m,p) is updated by:

$$\text{PCum } (m,p) \doteqdot M\Delta(m)$$

The process of the Jvm directly updating a per processor metric variable is now complete.

With reference to **Figure 29**, a flowchart depicting a process for the operating system kernel updating the value of a per processor metric variable in response to a request from the Jvm in accordance with a preferred embodiment of the present invention. As mentioned above,

0
D
O
C
U
M
E
N
T
S
:

not all operating systems provide a predefined per processor data area for storing processor related data. Those operating systems must allocate data areas for storing processor related information, such as, the value of per processor metric variables, $Pcum (M,p)$, for example. In this case the Jvm is not able to directly access the operating system kernel allocated per processor data areas. Therefore, the Jvm must rely on the operating system kernel for updating the value of per processor metric variables stored in those data areas.

The process begins with the Jvm recognizing that a metric (m) being tracked for a current thread (t) has changed by an amount represented by $MDelta (m)$ (step 2902). The Jvm then requests the operating system kernel update the metric (m) in response to $MDelta (m)$ (step 2904). Finally, the operating system kernel ensures that the value of the metric variable, $PCum (M,p)$, is updated without releasing the current processor (step 2906). The operating system kernel knows which processor the current thread (t) is running on. The operating system kernel then accesses the data area for that processor and updates the value of the accumulated per processor metric variable, $PCum (m,p)$, using the value of the change in the metric variable $MDelta (m)$. The metric variable $Pcum (m,p)$ is updated by:

$$PCum (m,p) += MDelta (m)$$

The process of the operating system kernel updating a per processor metric variable in response to a request from the Jvm is now complete.

With reference to **Figure 30**, a flowchart depicting the operating system kernel updating a metric variable value in response to a thread dispatch event in

accordance with a preferred embodiment of the present invention. Each time a new thread is dispatched, the value of the per processor metric variable for the old thread running on processor (p), TCum (M,p) and maintained in processor (p)'s data area, must be used to update the value of the thread accumulated metric variable for thread (t_{old}), TCum (M, t_{old}), held in the operating system kernel's data area. Thereafter, the value of the last accumulated metric variable for processor (p), PLastCum (M,p), which will be used to start new thread (t_{new}), is replaced by the current value of the accumulated metric variable, PCum (M,p), being held in processor (p)'s data area. Processor (p) can then proceed in processing new thread (t_{new}) while accurately tracking metrics for the new thread (t_{new}).

The process begins when processor (p) determines that a thread switch is necessary. Thread (t_{old}) on processor (p) is stopped and thread (t_{new}) is dispatched to processor (p) (step 3002). The operating system kernel then calculates the value of the change in metrics (M), Delta (M) (step 3004). The value of the change in metric, Delta (M), is found by decreasing the value of the accumulated metric variable for processor (p), PCum (M,p), by the value of the last accumulated metric variable for processor (p), PLastCum (M,P). The metric variable Delta (M) is calculated by:

$$\text{Delta (M)} = \text{PCum (M,p)} - \text{PLastCum (M,p)}$$

Next, the operating system kernel resets the value of the last accumulated metric variable for processor (p), PLastCum (M,p) (step 3006). The value of the last accumulated metric variable for processor (p), PLastCum (M,p), is replaced by the current value of the

accumulated metric variable for processor (p), PCum (M,p). The metric variable PLastCum (M,p) is reset by:

$$\text{PLastCum (M,p)} = \text{PCum (M,p)}$$

Finally, the operating system kernel updates the value of the accumulated metric variable for thread (t_{old}), TCum (M, t_{old}) (step 3008). The current value of TCum (M, t_{old}) held in the operating system kernel's data area must be increased to accurately reflect the change in the value of the metric, Delta (M), which was acquired from the metric values held in processor (p)'s data area, as described in step 3004 above. The metric variable TCum (M, t_{old}) is updated by:

$$\text{TCum (M, } t_{old}) += \text{Delta (M)}$$

The process is now complete for updating a metric in response to a thread being dispatched to processor (p).

With respect to **Figure 31**, a flowchart depicting the process for updating metric variable values in response to method entry or exit event in accordance with a preferred embodiment of the present invention. The process begins with the profiler receiving a entry or exit event for a method on thread (t) from the Jvm (step 3102). The profiler then requests the current value of the accumulated metric variable for thread (t), TCum (M, t) from the operating system kernel (step 3104). In response to the request from the profiler, the operating system kernel calculates the value of the change in metrics (M), represented by Delta (M), from processor (p) which is currently running thread (t) (step 3106). The value of the change in metrics (M), Delta (M), is the difference between the present per processor value of the accumulated metric variable for processor (p) was processing thread (t), PCum (M,p), and the per processor

value of the last accumulated metric variable for that processor, PLastCum (M,p). The metric variable Delta (M) is calculated by:

$$\text{Delta (M)} = \text{PCum (M,p)} - \text{PLastCum (M,p)}$$

Having used the value of PLastCum (M,p) for calculating Delta (M), the kernel resets PLastCum (M,p), to the current value of the accumulated metric variable on processor (p), PCum (M,p) (step 3108). The metric variable PLastCum (M,p) is reset by:

$$\text{PLastCum (M,p)} = \text{PCum (M,p)}$$

The operating system kernel then updates the value of the accumulated metric variable for thread (t), TCum (M,t) (step 3110). The metric variable TCum (M,t) is held in the operating system kernel's data area and must be increased by the value of the change in the metric variable on processor (p), Delta (M), in order to reflect the change in the value of metrics (M) attributed to processor (p). The metric variable TCum (M,t) is updated by:

$$\text{TCum (M,t)} += \text{Delta (M)}$$

Next, the value of the accumulated metric variable for thread (t), TCum (M,t), is sent to the profiler. The profiler can now calculate the change in metrics (M), ProfDelta (M), due to the new accumulated metric variable for thread (t) (step 3112). The change in metrics (M), ProfDelta (M), can be determined from the difference in the value of the accumulated metric variable, Tcum (M,t), received from the operating system, and the value of the accumulated metric variable for thread (t), ProfCum (M,t), held by the profiler. The metric variable ProfDelta (M) is calculated by:

$$\text{ProfDelta (M)} = \text{Tcum (M,t)} - \text{ProfCum (M,t)}$$

Next, the profiler uses the change in metrics (M), ProfDelta (M) to update the base value of the current method metric variable, Method (M, t). Method (M, t) updates the base value of the current method by:

Method (M, t) += ProfDelta (M)

Finally, the profiler sets the old value of the accumulated metric variable for thread (t), ProfCum (M, t), held by the profiler with the value of the accumulated metric variable for thread (t), Tcum (M, t), received from the operating system kernel. The metric variable ProfCum (M, t) is set by:

ProfCum (M, t) = Tcum (M, t)

The process for the profiler updating a base metric variable in response to method entry or exit event is now complete.

It is important to note that while the present invention has been described in the context of a fully functioning data processing system, those of ordinary skill in the art will appreciate that the processes of the present invention are capable of being distributed in a form of a computer readable medium of instructions and a variety of forms and that the present invention applies equally regardless of the particular type of signal bearing media actually used to carry out the distribution. Examples of computer readable media include recordable-type media such as a floppy disc, a hard disk drive, a RAM, and CD-ROMs and transmission-type media, such as digital and analog communications links.

It is important to note that while the present invention has been described in the context of a single active Jvm within an operating system, there are no constraints to its application to multiple Jvms. This

generalization is well within the means of those with ordinary skill in the art.

The description of the present invention has been presented for purposes of illustration and description, but is not limited to be exhaustive or limited to the invention in the form disclosed. Many modifications and variations will be apparent to those of ordinary skill in the art. For example, the present invention may be applied to other interpreted programming systems and environments other than Java. The embodiment was chosen and described in order to best explain the principles of the invention the practical application and to enable others of ordinary skill in the art to understand the invention for various embodiments with various modifications as are suited to the particular use contemplated.

01264.234-0 01265000